NEW DESIGN AND IMPLEMENTATION OF THE MANAGER-CLIENT PAIRING FRAMEWORK IN MANIFOLD

A Thesis
Presented to
The Academic Faculty

by

Roberto Pereira

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Computer Science in the
School of Computer Science

Georgia Institute of Technology
December 2013

NEW DESIGN AND IMPLEMENTATION OF THE MANAGER-CLIENT PAIRING FRAMEWORK IN MANIFOLD

Approved by:

Dr. Tom Conte, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
School of The School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: 12/11/2013

ACKNOWLEDGEMENTS

I would like to give special thanks to my parents who have given me all the support and encouragement I need to pursue my goals. I would like to thank Dr. Conte and Dr. Yalamanchili for advising me and providing feedback for my project.

I would also like to thank the members of the TINKER group: Eric Hein, Brian Railing, Rishiraj Bheda, Jason Poovey and Phillip Vassenkov for everything they taught me and for helping me throughout my project. Lastly, I would like to thank Dr. Richards and Dr. Shetty for helping me prepare my proposal and thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

MCP	Manager-Client Pairing
API	Application Programming Interface
ISA	Instruction Set Architecture

SUMMARY

Researchers in the area of computer architecture rely very heavily on the use of architectural simulators in order to perform their work. However, creating these simulators can be a very costly and time-consuming task. For this reason, the Manifold framework was created as an effort to streamline the creation of architectural simulators. The framework consists of a simulation kernel and several modules that model the different components of a system, such as the processor or memory. One of the modules currently present in the simulator, the mcp-cache module, models a cache system that implements the Manager-Client Pairing (MCP) framework. The MCP framework provides a scalable interface for creating coherence hierarchies that may use different coherence protocols throughout the hierarchy. However, the current implementation of the mcp-cache module is restricted to a two level cache hierarchy and doesn't accurately model the MCP framework.

This document presents the redesign and new implementation of the mcp-cache module in Manifold. The new module accurately models the MCP framework, allows the creation of cache hierarchies of arbitrary sizes and can be extended to support additional coherence protocol. The creation of the new module will enable the study of how different cache hierarchies, which vary in size and coherence protocols, perform when processing the same workloads. Additionally, the new module will be useful to test changes that could be made to the MCP framework in order to improve its performance.

CHAPTER 1

INTRODUCTION

Creating new hardware prototypes takes a long time and is very costly. For this reason, researchers in the area of computer architecture frequently resort to using architectural simulators to investigate their new ideas. However, creating a new simulator for each specific study requires a considerable amount of time and is very wasteful since much of the code necessary to build a simulator can be common to all simulators regardless of their specific purpose. The Manifold framework [1] is a project aimed at simplifying the creation of architectural simulators by allowing the reuse of common code and providing a simple way to change or add, in the form of modules, code that is dependent on the purpose of a given simulator.

This document discusses the redesign and implementation of a cache module in the Manifold framework. The new module fully implements the Manager-Client paring (MCP) interface, a composable interface for coherence hierarchies, and is designed to be easily extended to support the addition of new coherence protocols. This new module will provide a platform to study how MCP works with different protocols in coherence hierarchies of arbitrary width and height and provides a way to test potential improvements to the MCP interface. The rest of this chapter provides an introduction to both, Manifold and MCP and discusses the problems with the current cache module implementation. Chapter 2 presents and explains the technical considerations taken when implementing MCP and then provides an in-depth discussion of the implemented design. Lastly, Chapter 3 contains concluding remarks and suggested future work.

Manifold: A multicore modeling and simulation framework

The Manifold project is an open source effort by various groups at the Georgia Institute of Technology to create a scalable framework that facilitates the modeling and simulation of parallel systems architectures [1]. Manifold is not a simulator itself, but a framework that streamlines the process of creating architectural simulators targeted for the study of a particular area of a system. The main goal of the framework is to allow for full system simulation while maintaining the flexibility to change different components in the system, such as the processor or memory, without having to change any of the others. To this end, the framework is designed in two different layers: the kernel layer and the model layer. The kernel layer is responsible for all the simulation functionality such as event handling and process communication. The model layer consists of models of the architectural components in the system that integrate with the kernel layer to communicate with each other [2]. These models can be any of the ones already present in Manifold or new ones defined by the user. One of the models provided in Manifold is MCP-cache, which models a cache that implements the Manager-Client Pairing interface described in the next section. The separation of layers and models allows the user to try different models for a component without having to worry about the internals of the other components or the simulation kernel.

In order to create a simulator using the Manifold framework the user just needs to make a series of appropriate kernel Application Programming Interface (API) calls. A typical call sequence would: initialize the kernel, create the necessary components from the desired models, create and register with clocks, make the corresponding connections between components, set the simulation stop time and run the simulation. An additional call to the kernel is necessary for clean up after the simulation is completed [3].

Currently, simulators created with Manifold can be trace driven or execution driven. The trace driven simulators take as an input a trace following the PIN format. Execution-driven simulators use QSim as a front-end. QSim is a full system emulator that

provides a thread safe and ISA independent API [4]. QSim allows the simulators to be run sequentially using QSimLib or in parallel by using a QSim server.

Manager-Client Pairing: A Composable Interface for Coherence Hierarchies

The Manager-Client Pairing (MCP) framework was created as a way to unify the design of coherence protocols for multi-level cache systems. The framework provides a scalable cache coherence solution while allowing different coherence protocols to be used throughout the system. The MCP framework achieves its scalability and support for heterogeneity by formally defining an interface that limits the interaction between the different coherence protocols in the levels of the hierarchy. The interface enables the composition of coherence protocols to easily form a hierarchy [5]. Another benefit of this interface is that it greatly simplifies the task of verification. If all of the protocols that form part of a given MCP hierarchy are verified and meet the specification of the MCP interface, then the entire hierarchy is guaranteed to be verified as well [6].

The MCP framework defines two main entities: clients and managers. Managers manage read and write permissions for data, and clients hold these permissions. The framework also provides the notion of a coherence realm. A coherence realm consists of one manager and its associated clients. All the entities in a coherence realm share the same coherence protocol [5]. Figure 1.1 provides an illustration of a coherence realm.

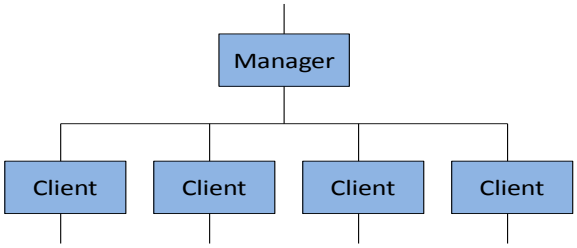

Figure 1.1 Coherence Realm

A coherence hierarchy is composed of coherence realms that communicate with each other through the MCP interface at the boundary of one realm's client and another realm's manager. Figure 1.2 illustrates a portion of such a hierarchy.

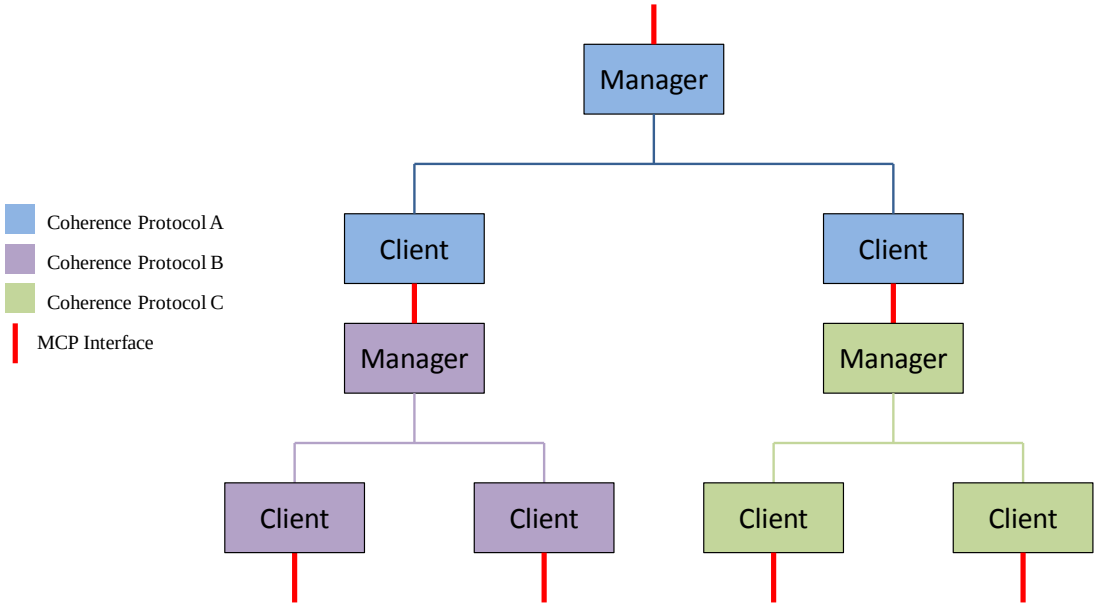

Figure 1.2. Coherence Hierarchy. Entities of the Same Color Communicate Over the Same Coherence Protocol.

The MCP interface is defined as a set of actions that a manager from a lower tier and a client from an upper tier can use to communicate with each other. Table A.1 in Appendix A lists the available MCP Actions. Figures 1.3 and 1.4 show an example scenario of how the MCP actions would be used in a MCP hierarchy when a processor issues a write request. Figure 1.3 shows the propagation of the permission requests and demands originating from the processor that issued the write request and Figure 1.4 shows the replies and acknowledgements propagating back to the requesting processor.

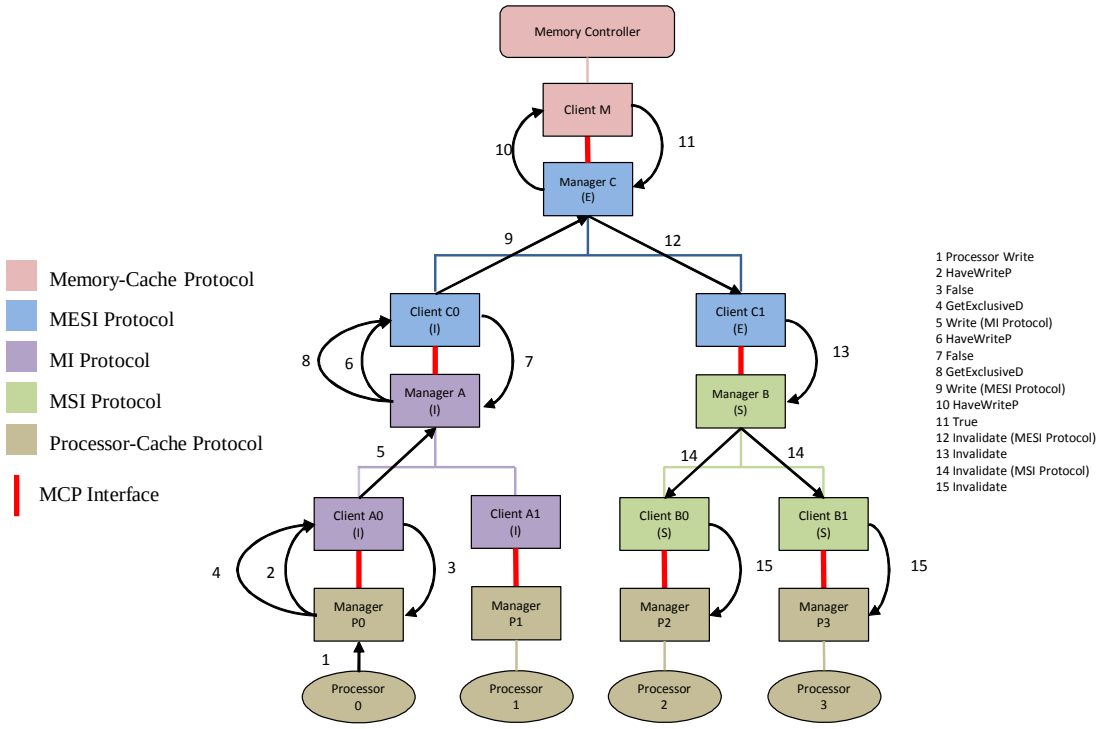

Figure 1.3. Propagation of Requests and Demands.

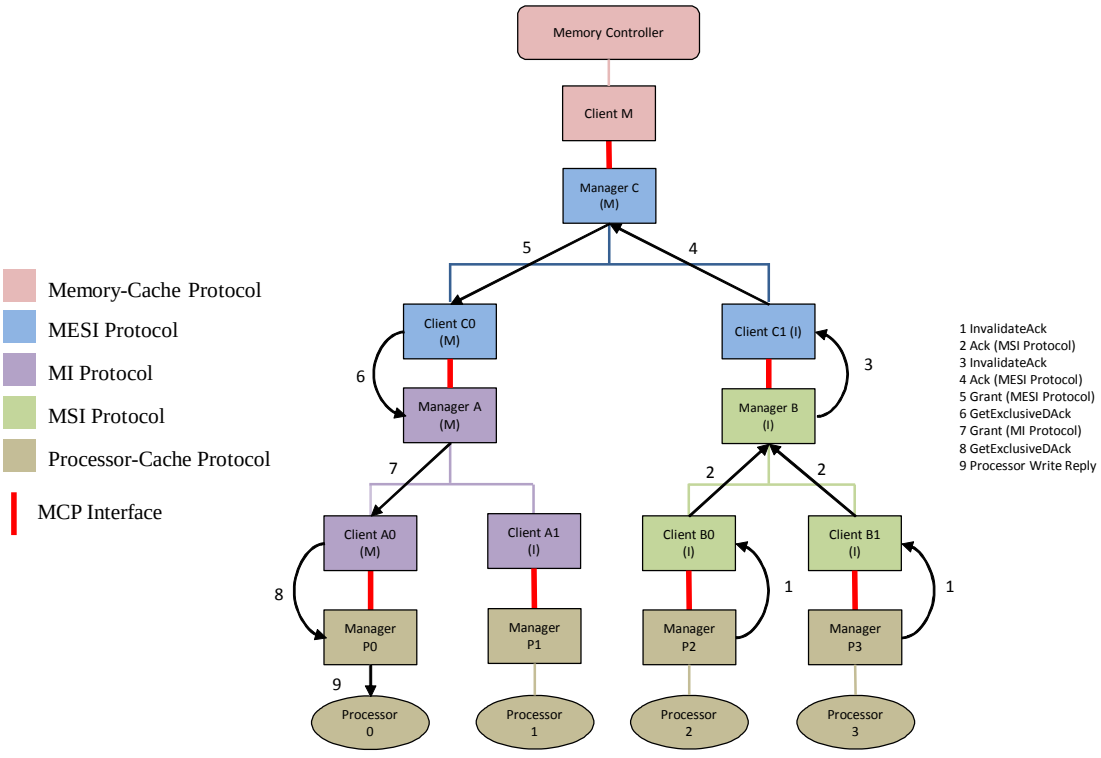

Figure 1.4. Propagation of Replies and Acknowledgements.

A textual description of the sequence of events depicted in Figure 1.3 is as follows. Processor 0 issues a write request to Manager P0 (1). Manager P0 queries Client A0, its paired client, for write permission with HaveWriteP (2). Client A0 doesn't have write permission since it is in the "I" state, so it returns false to Manager P0 (3). Manager P0 requests that Client A0 obtains write permission with GetExclusiveD (4). Client A0 issues a write request to Manager A using its native coherence protocol, MI (5). Manager A queries Client C0 for write permission with HaveWriteP (6). Client C0 doesn't have write permission, so it returns false to Manager A (7). Manager A requests that Client C0 obtains write permission with GetExclusiveD (8). Client C0 issues a write request to Manager C via its native coherence protocol, MESI (9). Manager C queries Client M for write permission with HaveWriteP (10). Client M has write permission, so it returns True (11). Manager C sends an invalidation coherence message to Client C1via its native coherence protocol, MESI (12). Client C1 demands that Manager B invalidates all of it clients that have read or write permission with Invalidate (13). Manager B issues an invalidation coherence message to Clients B0 and B1 using its native coherence protocol, MSI (14). Clients B0 and B1 demand that their managers invalidate all data with read or write permissions with Invalidate (15). Managers P2 and P3 are at the bottom of the hierarchy, so they invalidate their data and start the propagation of acknowledgements as shown in Figure 1.4.

Managers P2 and P3 signal to their paired clients that they have completed the invalidation of data with an InvalidateAck (1). Clients B0 and B1indicate that they have finished their invalidation process by sending an acknowledgment coherence message to Manager B via their native coherence protocol, MSI (2). Manager B signals to Client C1that it has completed invalidating with an InvalidateAck (3). Client C1 issues an acknowledgment coherence message to Manager C via its native coherence protocol, MESI (4). Manager C gives Client C0 write permission by issuing a grant coherence message to C0 using their native coherence protocol, MESI (5). Client C0 signals

Manager A that it has acquired write permission with GetExclusiveDAck (6). Manager A gives Client A0 write permission by issuing a grant coherence message to A0 using their native coherence protocol, MI (7). Client A0 signals Manager P0 that it has acquired write permission with GetExclusiveDAck (8). Manager P0 responds to the processor request by issuing a write reply.

This example illustrates a typical sequence of actions that occur in a MCP hierarchy. Requests originate at processors and propagate upwards in the hierarchy until they reach a level with sufficient permissions to service them. This level might initiate a series of demands that propagate downwards to make sure other protocols are updated as necessary. The responses to these demands will then propagate upwards back to the level that originated the demand. After all the replies have been gathered by the demanding level, the reply to the original request is propagated down to the requesting processor. An interesting property of the framework that can be observed in this example is the recursive nature of the interface, which greatly contributes to the scalability of the framework.

Issues with the Current implementation of Manager-Client Pairing in Manifold

The current mcp-cache module in the Manifold framework is intended to model a cache system that implements the Manager-Client Pairing interface. However, the current module implements a fixed two level cache hierarchy that doesn't make use of all the MCP actions. Figure 1.5 shows a diagram of the current module implementation. This implementation presents two main issues. The first issue with this implementation is that while it allows an arbitrary number of L1 nodes in the first level of the hierarchy (a variable hierarchy width), it doesn't support adding more levels, such an L3, to the hierarchy (a variably hierarchy height). This limitation makes it impossible to study the performance systems with hierarchies containing multiple coherence protocols.

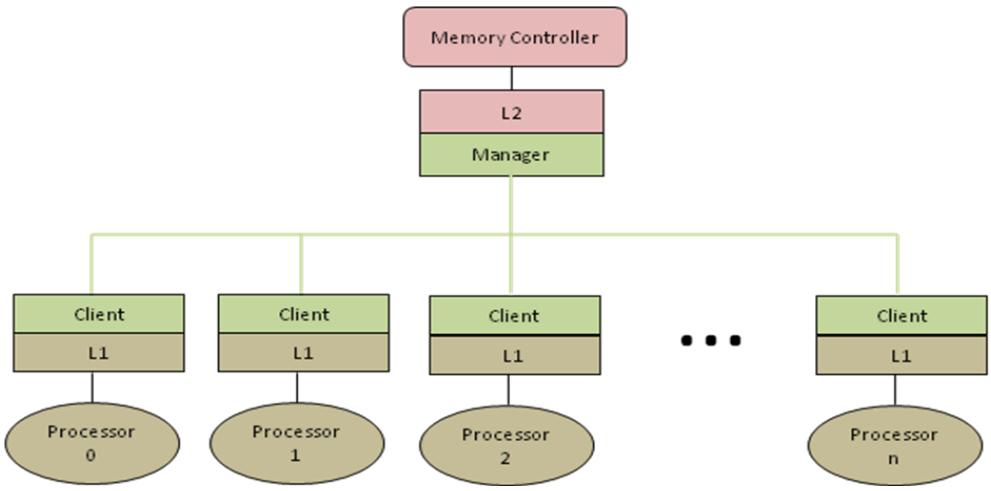

Figure 1.5. Current MCP Cache System.

The second issue with the current implementation of the cache system is that it doesn't make use of all the MCP actions. The only explicitly defined Manager and Clients are all part of the same coherence realm, so they communicate with each other using their native coherence protocol without the need to use the MCP interface. The L1 caches act as pseudo-managers by using MCP request actions to communicate with their clients, but the clients never use MCP acknowledgements to communicate back to the caches. Once a response is received at an L1 node, it is propagated directly to the processors. This lack of MCP acknowledgements makes the modeling of the MCP framework less accurate and, therefore, affects studies that might be carried out on the framework. The new implementation will address both of these issues by defining a new structure that makes it easy to create hierarchies of arbitrary depth and width and that will require the use of MCP requests and acknowledgements in every level of the hierarchy.

CHAPTER 2

NEW MANAGER-CLIENT PAIRING CACHE DESIGN

This chapter focuses on the design of the new implementation of the MCP cache module in Manifold. The first part of the chapter explains the technical aspects of the MCP framework that were taken into consideration when creating the new design. These aspects were those that significantly influenced the design of the new module but that are not readily apparent from the definition of the framework. The second part of the chapter explains the implemented design and discusses the reasoning behind various design decision taken. The overall goal of the design is to produce a module that accurately models a cache system implementing the MCP interface that is flexible, extensible and easy to use.

Influential Technical Considerations

Managers are Responsible for Data Storage

Managers at each level of the hierarchy, except for those at levels without data persistence, must keep track of the data that they have requested in order to satisfy future requests from both, clients in the manager's realm and clients from upper realms. There are two ways for managers to achieve this, as pictured in Figure 2.1. The manager can a) have storage of its own and store the data directly or b) interface with an external data store to which it can send data to and retrieve data from.

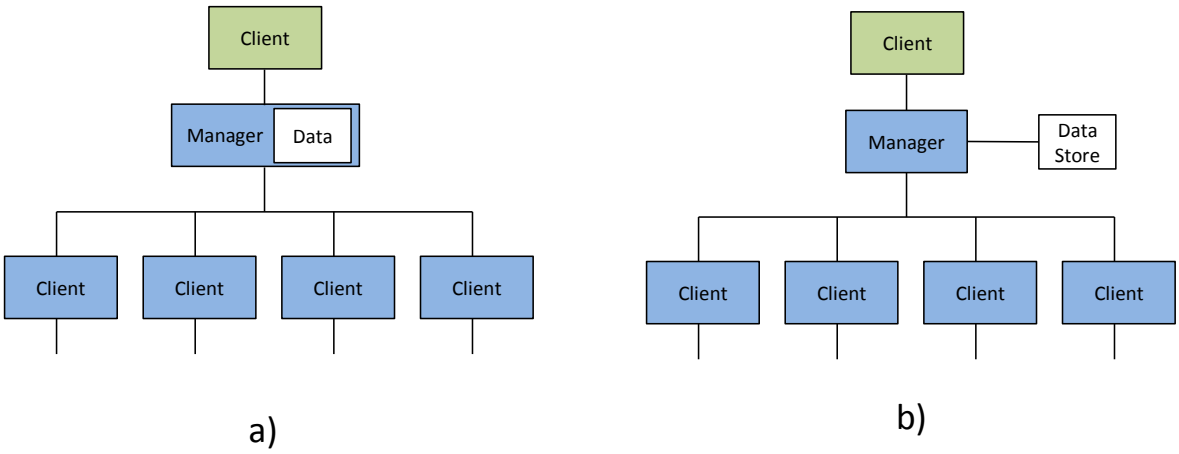

Figure 2.1. Manager's Alternatives for Maintaining Data.

The MCP actions that cause a manager to store data are GetReadDAck and GetExclusiveDAck, and the actions that cause a manager to supply data are Supply, SupplyInvalidate and SupplyDowngrade. Additionally, Invalidate and GetEvictAck cause a manager to delete its stored data. It is important to keep this responsibility of the managers in mind since it determines part of the resources that a manager needs, in case the data is stored locally, or what interfaces it will make use of to communicate with a data store, in case the data is stored externally.

Hierarchy Levels without Data Persistence

A very interesting property of the MCP framework is that it allows levels without data persistence in a hierarchy. These levels are only for coherence purposes and do not store any data besides the metadata necessary for coherence managing. The support of levels without data persistence makes it possible to map a single coherence hierarchy to many different cache hierarchies. For example, consider the coherence hierarchy shown in Figure 2.2.

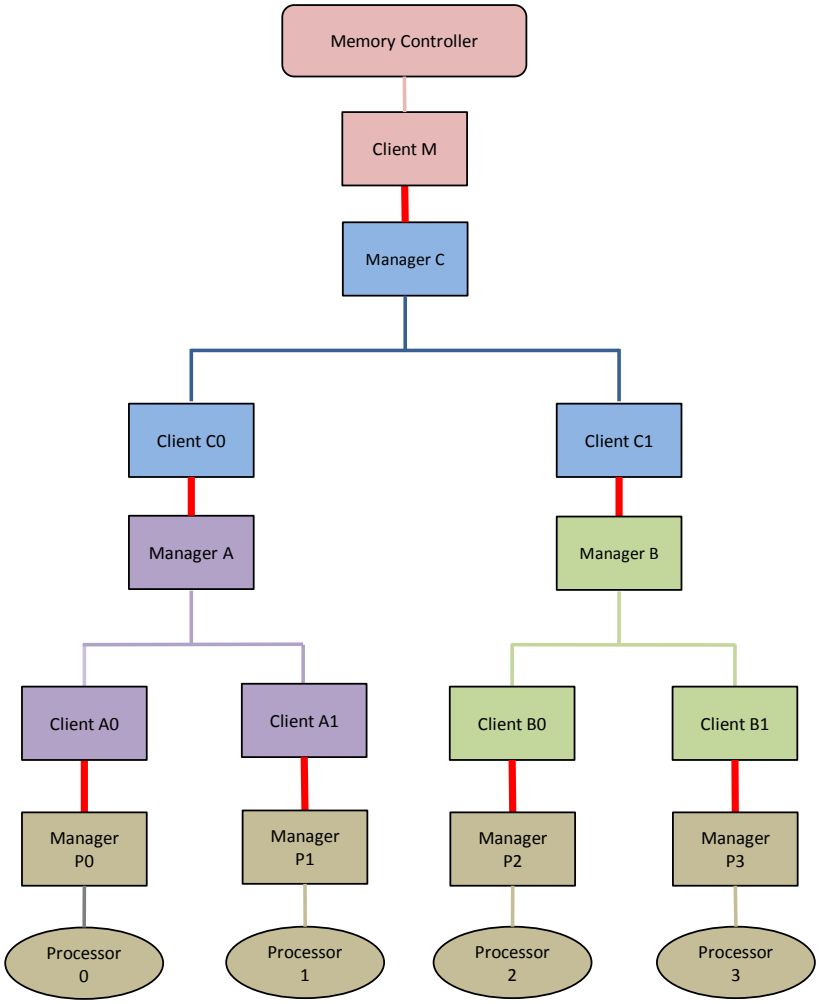

Figure 2.2. Coherence Hierarchy with Up to Three Different Coherence Protocols.

The most intuitive way to map this three-level coherence hierarchy to a cache hierarchy is by having a three-level cache hierarchy where each manager-client pair in the coherence hierarchy is associated with a cache of the same level in the cache hierarchy. The pairs containing managers P0, P1, P2 and P3 would be associated with an L1 cache, the pairs containing managers A and B would be associated with an L2 cache, and the pair containing manager C would be associated with an L3 cache. These

associations are denoted in the hierarchy shown in Figure 2.3. In this hierarchy all levels have data persistence.

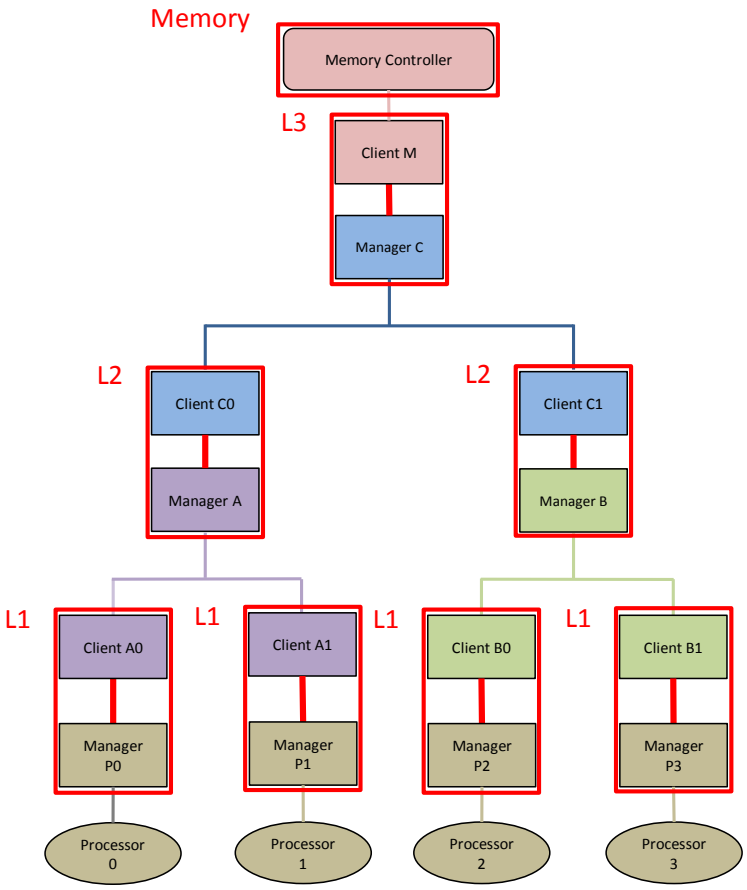

Figure 2.3. Possible Three-level Cache Hierarchy

Another way to map the coherence hierarchy in Figure 2.2 to a cache hierarchy is shown in Figure 2.4. This cache hierarchy is similar to the one in Figure 2.3, but the L3 cache has been removed and the manager-client pair associated with it has been moved to memory. In this case manager C's level doesn't have data persistence and only stores coherence information similarly to a directory that is located before the memory controller. The same approach can be taken one step further by removing both L2 caches

and moving their associated manager-client pairs to memory to create yet another possible cache hierarchy for the same coherence hierarchy as shown in Figure 2.5.

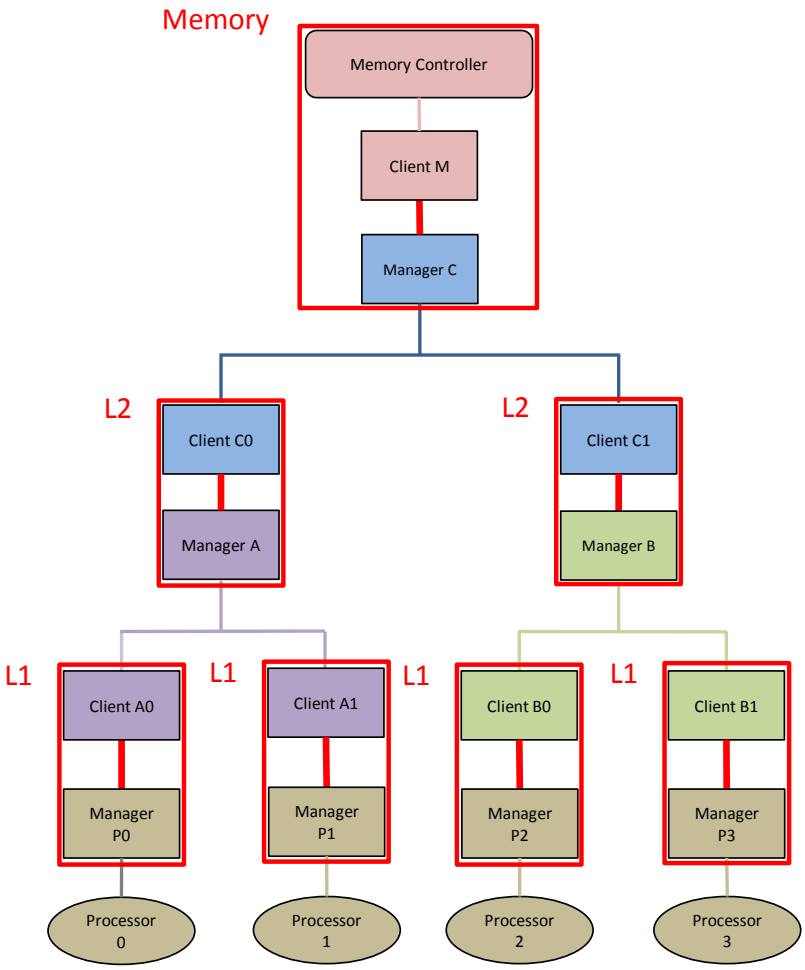

Figure 2.4 Possible Two-Level Cache Hierarchy

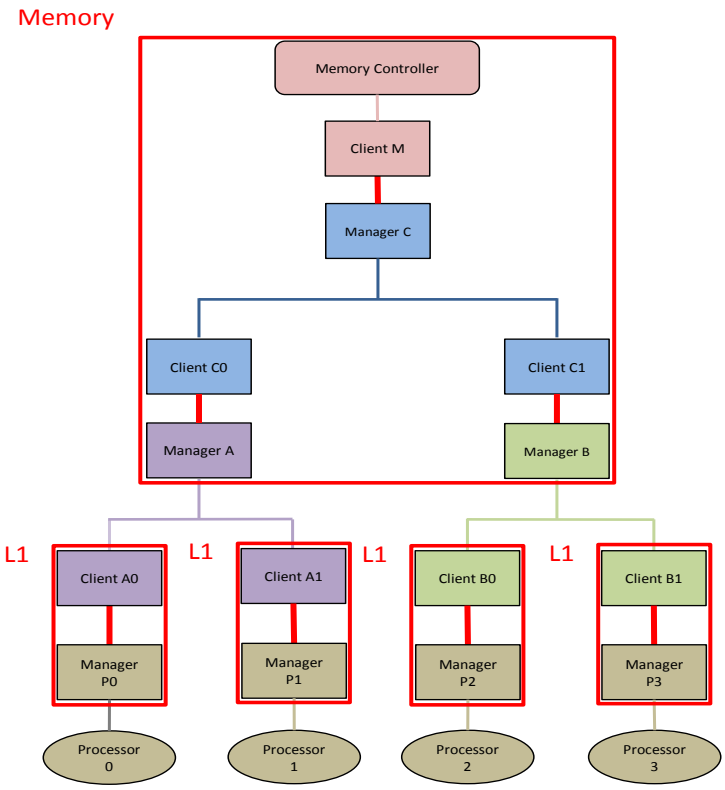

Figure 2.5 Possible One-Level Cache Hierarchy

Hierarchies such as the ones shown in Figures 2.4 and 2.5 would be more desirable than the one shown in Figure 2.3, if there is a need for the use of different coherence protocols within a system, but not enough levels of caching are available to support data persistence at all levels. If a coherence hierarchy has more levels than the cache hierarchy it is associated with, then there will be coherence levels without data persistence.

The existence of levels without data persistence is very influential to the implementation of the framework since different coherence messages and MCP actions are used based on whether or not a level has data persistence. For instance, consider the example illustrated by Figures 1.3 and 1.4. In this example, it was assumed that all levels

of the hierarchy had data persistence. If only the first level of the hierarchy had data persistence (Managers P0, P1, P2 and P3) then the series of requests and demands for this scenario would be as shown in Figure 2.6 and the series of replies and acknowledgements would be as shown in Figure 2.7. The main difference between these two scenarios happens at Manager C. In the first case, where there is data persistence, Manager C is able to supply the data to Client C0, so it just sends an invalidation message to Client C1 that propagates downwards. In the second case, where there isn't data persistence, Manager C is not able to supply the data to Client C0, so it has to request Client C1 not only to invalidate, but also to supply the manager with data. This additional supply request propagates downwards and may cause other supply requests to be generated depending on whether or not the lower levels have data persistence. In this case Manager B also doesn't have data persistence, so it has to ask one of its clients to supply the data. From these scenarios it is clear that the behavior of a manager is dependent on whether it is located in a level with data persistence or not.

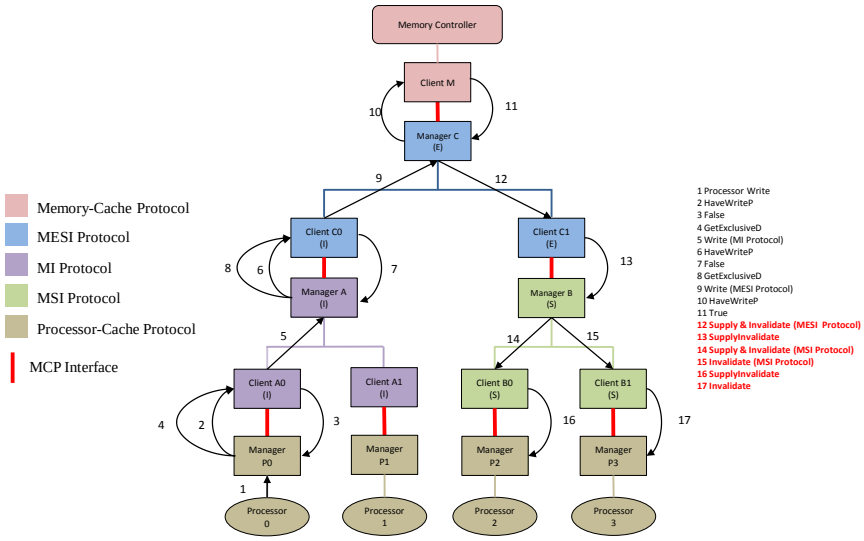

Figure 2.6. Propagation of Requests and Demands in a Hierarchy Containing Levels without Data Consistency.

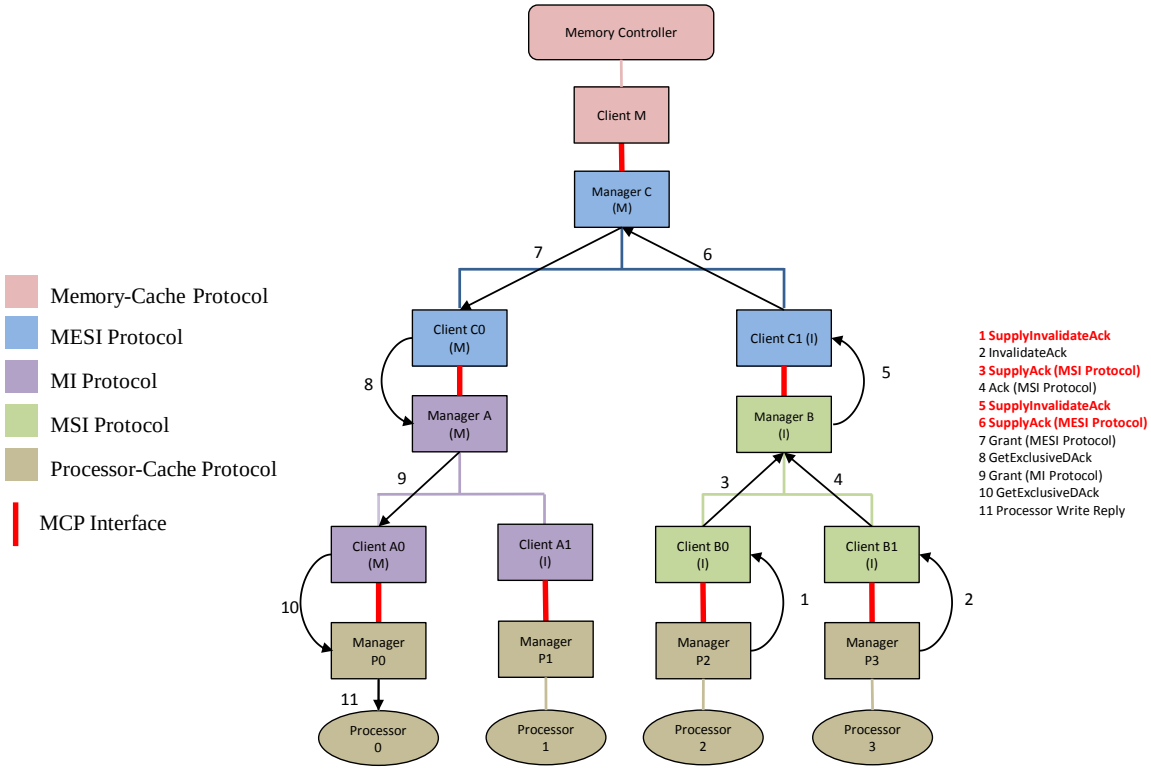

Figure 2.7. Propagation of Replies and Acknowledgements in a Hierarchy Containing Levels without Data Consistency.

Hierarchies with Distributed Caches

Another important aspect to consider is how the MCP framework operates in cache hierarchies with distributed levels. In distributed levels, each slice of the distributed cache has its own manager and each of these managers is connected to all the clients that are serviced by the distributed cache. When a client from a lower level, serviced by a distributed cache, needs to process a request, it sends a coherence message to the appropriate manager based on the address of the request. The manager that receives this

request doesn't need to be aware that it is part of a distributed level and it can just process the request it received using the available MCP actions. Figure 2.8 shows an example scenario of a MCP cache hierarchy with distributed L2 caches servicing a read request from a processor.

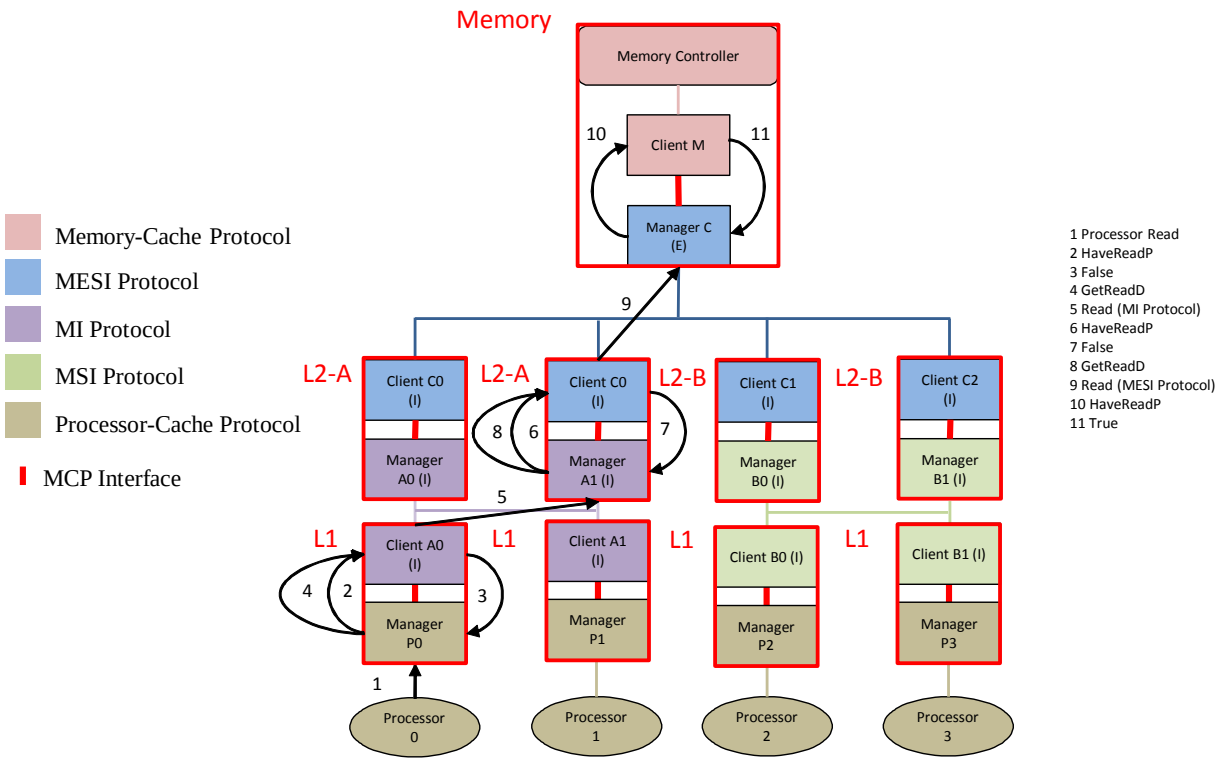

Figure 2.8. Propagation of Requests in a Hierarchy Containing Distributed L2 Caches.

In this figure we can observe a system with four processors, four L1 caches, two L2 caches and a memory system. The L2 caches are both distributed into two slices and each slice can store data for half of the total memory address space covered by the cache. The slices for the first L2 cache are denoted as L2-A and the slices for the second L2 cache are denoted as L2-B in the figure. In this example, processor 0 issues a read request that is processed as described in section 1.2 until it reaches Client A0. When the request

reaches Client A0, this client must send its coherence message to the appropriate L2-A slice based on the address of the request. In this case, the appropriate slice is the one containing Manager A1. After the request reaches Manager A1, it is processed normally by the upper levels of the hierarchy until it reaches Manager C, which starts the propagation of replies down the hierarchy (not shown in the figure). From this example, looking at the coherence realm using the MI protocol, it may seem like there are multiple managers for a coherence realm, but this is not the case. Managers A0 and A1 are responsible for a disjoint set of address so, in essence, they are just both parts of a distributed manager. Conceptually, it could be thought as if Clients A0 and A1 are being managed by Manager A which is distributed over two caches, as shown in Figure 2.9.

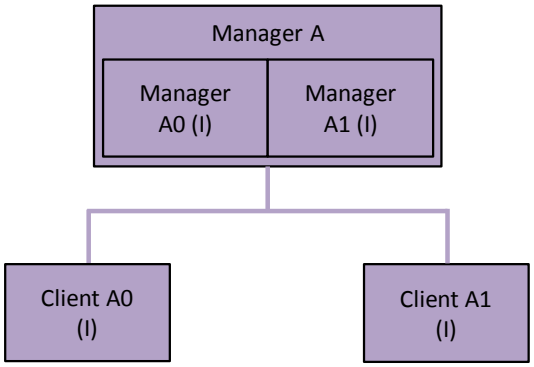

Figure 2.9. Unified View of a Distributed Manager.

This example hierarchy shows that in order to support distributed caches, the implementation of an MCP cache hierarchy must provide the necessary facilities to determine which manager a request should be sent to based on the address of the request.

New Design Overview

The new design for the MCP cache module in Manifold was created with three main goals in mind: accuracy, scalability and extensibility. The new design accurately models a cache system implementing the MCP framework by making use of all the MCP actions as described in the definition of the framework [5]. Furthermore, the design allows for the creation of cache hierarchies of arbitrary size and provides a simple interface that can be used to integrate new coherence protocols to be used in these hierarchies.

The design is based around a building block called an MCP Unit. An MCP Unit consists of a manager, a client and, optionally, a cache. A unit may not have a cache if it is part of a level without data persistence. The client and manager of an MCP Unit communicate with each other using MCP actions and if a cache is present the manager communicates with it for storage purposes. Additionally, an MCP Unit has two ports for external communication, the client port and the manager port. Figure 2.10 shows a graphical representation of an MCP Unit.

As shown in Figure 2.10, the managers in levels with data persistence will keep track of the data by communicating with an external storage, the cache. The decision to use an external storage instead of the manager storing the data locally was made for two reasons. First, having an external storage decouples the implementation of the manager and the data storage, if one of them has to be modified the other one is not affected. And second, this separation facilitates the creation of new manager implementations by allowing new managers to reuse an existing data store implementation without having to recreate it.

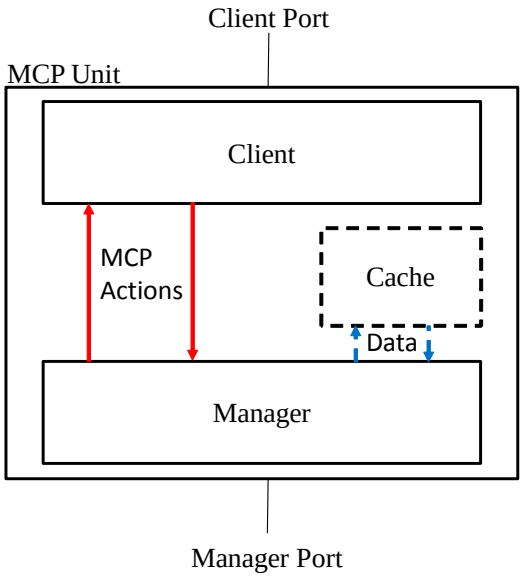

Figure 2.10. MCP Unit. Dashed Elements Are Only Present in Levels With Data Persistence.

The self-contained nature of the MCP Unit makes it an ideal building block for constructing cache hierarchies of arbitrary width and height. A complete system would consist of the desired number of MCP Units connected together, a memory controller connected to the upper most client of the root unit of the hierarchy and a processor connected to the manager of each of the leaf units of the hierarchy. Figure 2.11 illustrates one such hierarchy.

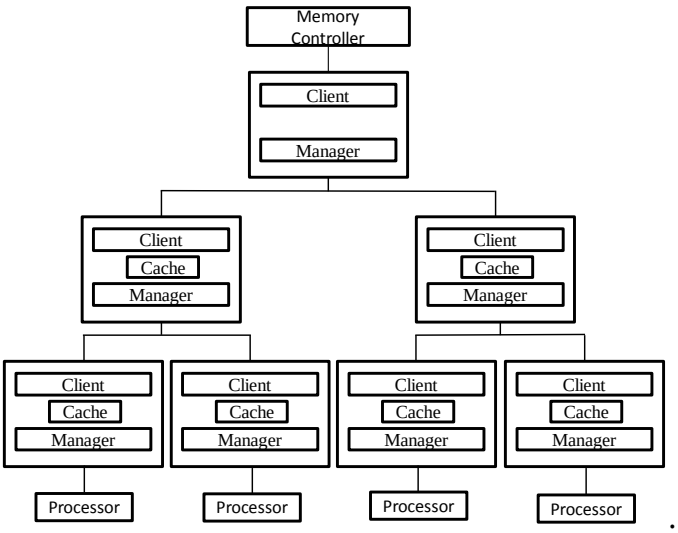

Figure 2.11. Cache Hierarchy Built with MCP Units.

This figure shows how to use six MCP Units to create a two-level cache hierarchy with a three-level coherence hierarchy. Note that the upper most MCP Unit doesn't have a cache since it is representing a level without data persistence. Lastly, the design makes it simple to add new coherence protocols by allowing the Manager and the Client in the MCPUnit to be changed. Only a new Client and a new Manager need to be created to implement a new protocol and not the entire unit.

Implementation Details of the New Design

The implementation of the new cache module consists of five main classes: MCPUnit, Cache, Manager, Client and CacheRequest. The MCPUnit class serves as a container for the other classes and is the point of integration between the rest of the module and Manifold. The Cache class provides storage functionality for the Manager class as needed. The Manager class models the behavior of a Manager in a Manager-Client pair by providing the demand a request reply MCP actions from Table A.1 in appendix A. Additionally, the Manager Class interacts with the Cache class when it needs to store or supply data. The Client class models the behavior of a Client in a Manager-Client pair by providing the permission query, permission get and demand reply MCP actions from Table A.1 in appendix A. The Manager and Client classes are provided as pure abstract classes that the user must derive from in order to implement protocol specific managers and clients. Lastly, the CacheRequest class is used as a message type that can be used for inter and intra MCPUnit communication. Figure B.1 in appendix B shows the class diagram for the new cache module in Manifold.

MCPUnit

The MCPUnit contains references to a Manager, a Client and a Cache and provides accessors for each one of them. MCPUnit is the only class that interacts directly with the rest of the Manifold framework. The class inherits the Component class from Manifold meaning that it can be instantiated as a component of a simulator and it can

communicate with other Manifold components using the Send function. The class defines an enumeration that numbers its available ports, the client port and the manager port, and provides a function handler for each port that handles messages received from the Manifold kernel. Since the messages delivered by the manifold kernel are of an unknown type to the MCPUnit, these handlers call auxiliary functions in their respective entities, the manager or the client, that convert the message to a CacheRequest which is then stored in a queue for processing. The unit maintains two queues of arriving requests, one for the requests arriving on the client's port and another for the requests arriving on the manager's port. These requests arrive asynchronously at the MCPUnit and are later processed when the RisingTick function is called. The RisingTick function represents a clock cycle and is called by the Manifold kernel periodically based on the simulation parameters. When the RisingTick function is called, the unit attempts to process at most a fixed number of requests. This limit on the number of requests to be processed per clock cycle is given as a parameter to the unit when it is created. The unit tries to process requests that may be stalled in the Cache and then tries to process requests from the queues of incoming requests from the network ports. Note that every time a requests is processed, it might make other requests that are stalled in the Cache available to be processed, so the order in which requests are checked to be processed must be maintained. For example, if there are two requests to be processed and it is determined that the first request must come from the incoming requests queues, this doesn't guarantee that the following request must also come from the same queues. The first request might have made a request on the Cache stalled queue ready to be processed so the second request to be proceed should come from this queue.

Lastly, MCPUnit provides the SendMessage function to be used by its Manager and Client to send messages to other components. This function provides an optional delay parameter in case the receiving component is connected directly, as opposed to being connected through a network, to the unit and the transmission time is known. The

SendMessage function determines the appropriate port to send the message on and then calls the Send function from the Component base class that handles inter component communication. The header file for the MCPUnit class can be found in Figure C.1 of appendix C. Additionally, the Manifold Component Developer's Guide [7] provides further information about Manifold components.

Cache

The Cache class is responsible for the data storage. It contains a hash table to use as a tag store, a map to keep track of miss status handling registers (MSHRs), a list of stalled requests and a reference to the Manager it is associated with. Notice that the hash table only stores the tag of addresses and not any actual data since it is not necessary due to the nature of the simulation. Furthermore, the implementation of the MSHRs is a very basic version of the one described by Kroft [8] and it is aimed at modeling the effects of having a number of pending cache requests that is limited by hardware resources. The map's keys are block addresses that map to the cache request that is currently occupying a given MSHR, and the number of keys in the map will be at most the number of MSHRs that the cache has. Requests that arrive to the Cache when all MSHRs are already occupied are stalled immediately.

The Cache provides six publicly accessible functions. GetHitTime and GetLookUpTime which return the hit time and the lookup time of the cache respectively. These functions will be called by the Manager to determine the delay to be used when sending a message. The function HasBlock returns true if a given block is present in the cache and false otherwise. AllocateBlock is used to reserve space in the cache for a given block address, If the cache is full, the function tries to evict another block to create space. If a block is successfully allocated, the function returns true, otherwise it returns false, meaning that the eviction could not complete due to the victim being busy or the cache not having the necessary permission to evict it. The ReleaseMSHR function is used by the Manager to signal to the cache that a request has been fully processed and no longer

needs an MSHR. This signal allows the cache to free the MSHR previously occupied by the completed request so that other requests may use it. Lastly, the ProcessStalled function attempts to process the next request from the stalled request list that is ready to be processed. If no such request exists the function returns false, otherwise it returns true. This function is called by the MCPUnit every clock cycle.

The list of stalled requests in the cache is used to hold requests that are waiting to be processed and there are four reasons for which a request might be stalled. A request will be stalled if: it can't obtain an MSHR, it requests a block and the cache needs to evict but the victim is busy or the cache needs to obtain permission before evicting, or if there is a previously stalled request for the same address. Each stalled request is assigned a status flag which determines if the request will be processed the next time the ProcessStalled function is called. This status flag is initialized to "not ready" and is set to "ready" when the event that a request is waiting for has occurred. For example, the status of a stalled request might change if an MSHR is released or an eviction is completed. ProcessStalled will process the first stalled request that has a ready status. It is important to note that requests are removed from the stalled list independently of the order in which they arrived to the cache, so a request that arrived later might be processed before one that arrived earlier. This reordering of read and write requests effectively implements a weak memory consistency model [9]. The header file for the Cache class can be found in Figure C.2 of appendix C.

Manager

The Manager class is provided as an abstract interface and is intended to be used as the starting point for users that want to add a new coherence protocol to the cache module. The class contains pointers to its associated MCPUnit, Cache and paired Client and also provides the private function SendPacket which can be used to send messages to another component after a specified delay. The rest of the class consists of pure virtual functions that must be implemented by the inheriting class to provide functionality since

the implementation of these functions depends on the coherence protocol that is being used. The function ConvertPacket converts a network packet into a CacheRequest and is called by the MCPUnit whenever a new packet arrives to the unit on the Manager port. HandleRequest takes in a CacheRequest and processes it. This function is called by the associated MCPUnit every clock cycle where there aren't any stalled requests ready to be processed in the Cache. The functions CanEvict and GetEvict query for and request eviction permissions respectively. These functions are called by the associated Cache when an eviction is needed. The rest of the functions in the class implement the demand and request reply MCP actions from Table A.1 in appendix A and are called by the paired Client when permissions are granted or when invalidating or supplying is necessary. The header file for the Manager class is shown in Figure C.3 of Appendix C.

Client

The Client class is also an abstract interface and it is very similar in structure to the Manager class. It contains pointers to its associated MCPUnit and paired Manager. The ConvertPacket and HandleRequest functions are again present and provide the same functionality as they do in the Manager. The CanEvict and GetEvict functions are not necessary in the Client as they are in the Manager since the Client is not responsible for data storage. The rest of the functions in the Client implement the permission query, permission get and demand reply MCP actions from Table A.1 in appendix A and are called by the paired Manager when permissions are requested or when an invalidation or supply has completed. The header file for the Client class in shown in Figure C.4 of Appendix C.

With these two interfaces, the Client and the Manager, it is very simple to add new coherence protocols to the module. The user only needs to create a new protocol-specific Manager and a new protocol-specific Client by subclassing the corresponding base classes and then use these new classes appropriately when creating the MCPUnits for the coherence realm that will use the new protocol.

CacheRequest

The CacheRequest class is used as a common message type that all the previously described classes can use to communicate with each other. It contains an address, a memory operation type, a source id, a destination id and a flag which indicates if the message is a reply. However, this information may not be sufficient for all implementations of Clients and Manager. In this case, the user can create a new message type class by first inheriting from the CacheRequest class and then adding the new fields that are necessary. The header file for the CacheRequest class in shown in Figure C.5 of Appendix C.

Typical Call Sequence

Consider an MCPUnit that is part of a cache hierarchy and that is connected to a processor through its Manager port and to another MCPUnit through its Client port. One such unit would look the like those in the first level of the hierarchy shown in Figure 2.11. A typical function call sequence generated by a read request from the processor connected to the Manager's port is illustrated by the sequence diagram in Figure 2.12. In this scenario the request hits on the cache.

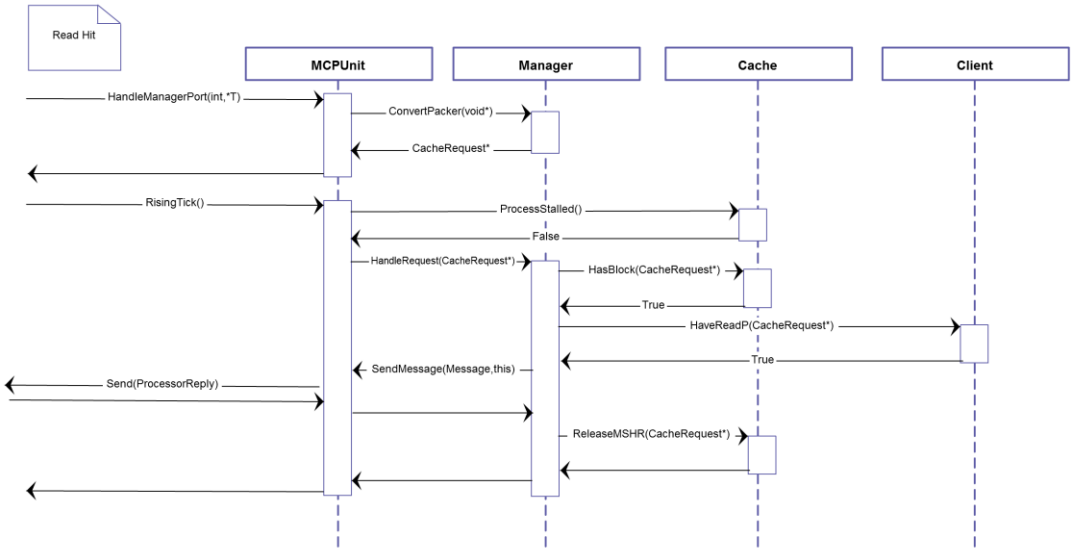

Figure 2.12. Sequence Diagram for a Read Hit.

First, the read request is handed by Manifold to the MCP unit via the manager's port handler function. This function passes the request to the Manager which converts it into a cache request and returns it to the MCP unit where it is queued for processing. At a later time, the Manifold kernel signals a clock cycle to the MCP unit. After receiving the clock signal, the MCP unit signals the cache to process stalled requests. Since the cache has no stalled requests it returns false to the MCP unit. The MCP unit then gets the read request from its internal queue and passes it to the manager for processing. The manager queries the cache to know if the block is present in the cache and since the block is present, the cache returns true. Afterwards, the manager queries the client for read permissions for which the client returns true. Since the block is present and has read permission, the processor request can be satisfied so the manager sends a reply to the processor by passing a message to the MCP unit. The MCP unit takes the reply from the manager and passes it to the Manifold kernel so it can be delivered to the processor. Lastly, since the processor request has been satisfied, the Manager signals to the Cache that it can release the MSHR that the request held.

In a case where the read request results in a miss instead of a hit, the sequence of events can be divided in two parts. The first part corresponds to the original request arriving at the manager and being queued to wait for the permission and data to be supplied from upper levels. Figure 2.13 shows the sequence diagram for this part. This diagram is similar to the one in Figure 2.12, but it diverges when the cache is queried for the presence of the block. In this case, the block is not present so the cache returns false to the manager when it is queried. The manager requests the allocation of a block to the cache which successfully allocates a block and returns true. After obtaining a block for the request the manager queries the client for read permissions. The client doesn't have the permission so it returns false. The manager then queues the cache request and indicates to the client to obtain read permission. The client sends a coherence request to his manager by passing a message to the MCP unit. Lastly, the MCP unit takes this

message from the client and passes it to the Manifold kernel so it can be delivered to the upper-tier manager. At this point the original processor read request is queued at the manager waiting for a response from the upper coherence realm to arrive.

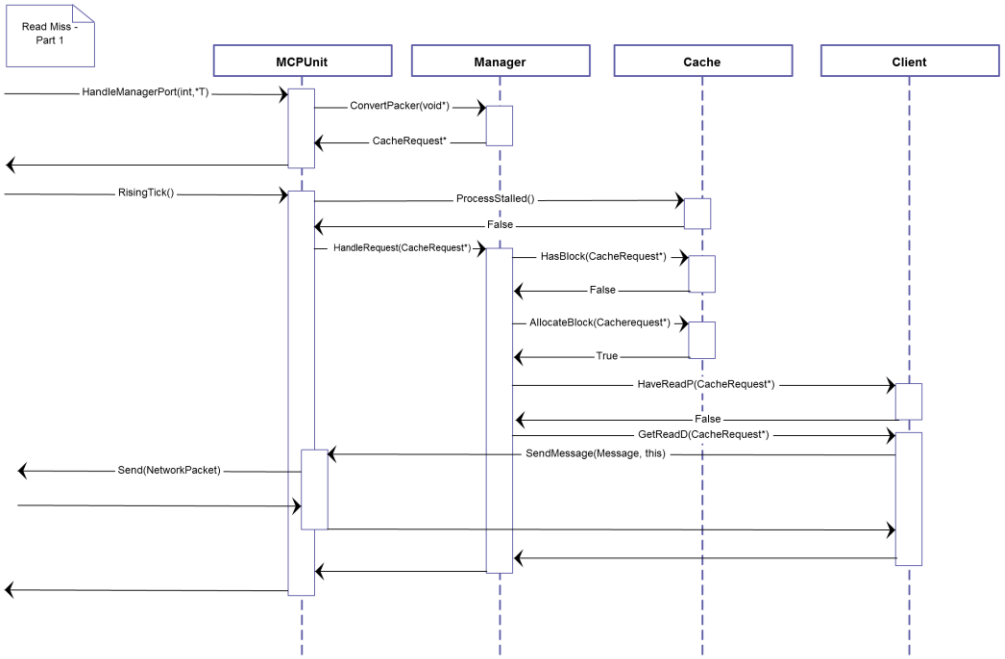

Figure 2.13. Sequence Diagram for Servicing a Read Request that Misses on the Cache and Is Stalled.

Figure 2.14 shows the second part of the sequence of events that is initiated when a reply to a read permission request arrives to the MCP unit from an upper realm.

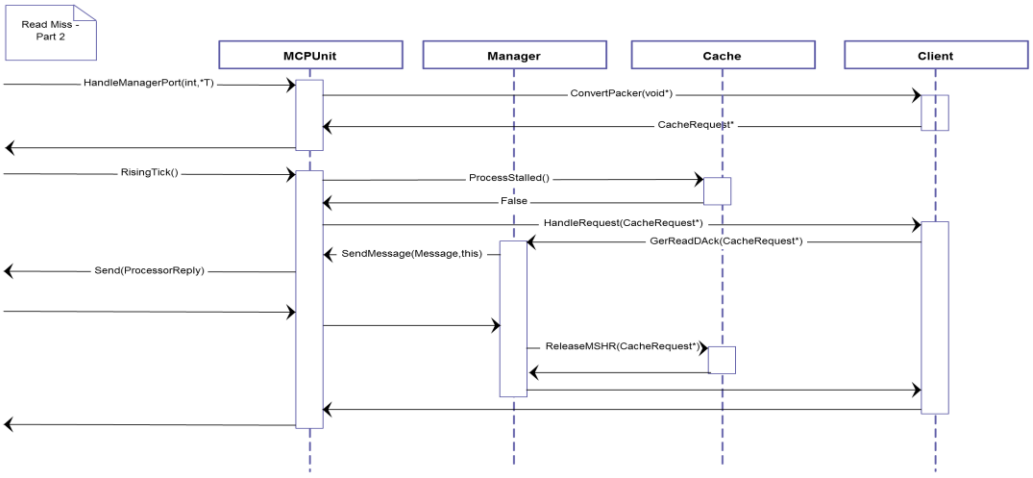

Figure 2.14. Sequence Diagram for Receiving Read Permission for a Stalled Request.

The sequence in this figure starts again with Manifold passing a message to the MCP unit, but this time the client's port handler function is used. This handler passes the message to the client which turns it into a cache request and returns it to the MCP unit where it is queued to be processed. When the next clock cycle happens, the Manifold kernel signals it to the MCP unit which starts processing requests. The MCP unit first signals to the cache to process its stalled requests but since there aren't any requests to be processed the cache returns false. The unit then gets the request from its queue and passes it to the client which signals the manager that read permission has been acquired for a particular address. After receiving the acknowledgment, the manager wakes up the request that was waiting for this reply. Since now both, the cache block and read permission are present, the manager can now satisfy the request and send a reply back to the processor. This reply is sent as before, the manager passes a message to the MCP unit, which in turn forwards it to the Manifold kernel, so it can be delivered to the processor. At this point, the processor request has been satisfied so the Manager signals the Cache to release the MSHR that was being used by the request.

These example scenarios only represent a small set of all the cases that can happen in the module. However, the general functioning of the module is the same for most cases. A message, request or reply, arrives to a unit. This message is then passed to the corresponding entity, the manager or the client, for processing. If the request can be processed immediately, it generates another message that is sent to the next appropriate realm. If the request can't be processed, it is queued either at the cache, for space reasons or at the manager or client, for coherence reasons. A request will be queued until it receives the appropriate reply that it is waiting for and then it continues processing. This process takes places in all the MCP units of the system and it continues until all the requests are satisfied.

CHAPTER 3

CONCLUSION

A redesign and a new implementation of the Manager-Client Pairing cache module for the Manifold framework have been presented. This new design models the Manager-Client Pairing interface more accurately than the current implementation, provides a simple way to create cache hierarchies of arbitrary height, width and coherence protocol composition and is readily extensible to support additional coherence protocols. With this new implementation it will be now possible to study the performance of different cache hierarchies that vary in width, height and coherence protocols. Furthermore, the new implementation also allows the testing of changes that might be made to the Manager-Client Pairing interface in order to improve its performance. Lastly, this document also serves as a reference for anyone wanting to modify the implementation of the module and for users of the framework.

Future Work

There is still work that could be done in order to further improve the cache module. A very useful feature to be included in the module would be the ability to describe coherence protocols as state machines in input files. This feature would eliminate the need for recompilation and greatly reduce the time in which new protocols can be added to the module. Another possible improvement would be adding a configuration variable that allows the user to select which consistency model to use amongst a list of possible choices. Lastly, the implementation of the miss-status holding

registers could be modified in order to model a more advanced scheme such at the one proposed by Farkas and Jouppi [10].

APPENDIX A

MCP ACTIONS

Lower Tier Manager to Upper Paired Client Permission Query

HaveReadP	Return true if paired Client has read permission
HaveWriteP	Return true if paired Client has write permission
HaveEvictP	Return true if paired Client can be safely evicted

Lower Tier Manager to Upper Paired Client Permission *Get*

GetReadD	Paired Client begins data and read permission acquisition sequence within it's native coherence realm. L1/Lower Manager expects GetReadDAck upon completion.
GetExclusiveD	Paired Client begins data and write permission acquisition sequence within it's native coherence realm. L1/Lower Manager expects GetExclusiveDAck upon completion.
GetExclusive	Paired Client begins write permission acquisition sequence within it's native coherence realm. L1/Lower Manager expects GetExclusiveAck or GetExclusiveDAck upon completion.
	Used when data is already available in L1/Lower Manager (HaveData == true) and only a permission upgrade is required.
	May be satisfied by a GetExclusiveDAck if upper tier protocol demands a downgrade while GetExclusive is in flight, causing HaveData to become false.
GetEvict	Paired Client begins eviction sequence within it's coherence realm. L1/Lower Manager expects GetEvictAck upon completion.
	Used when block ownership or most recent dirty version resides in L1/Lower Manager's realm.
	Needs to include data payload when data being evicted is dirty.

Upper Tier Client to Lower Paired Manager Permission Request Reply

GetReadDAck	Response by paired Client to complete previous GetReadD request. Supplies data packet and signifies paired Client (and thus lower Manager's realm) now has read permissions.
GetExclusiveDAck	Response by paired Client to complete previous GetExclusive/GetExclusiveD request. Supplies data packet and signifies paired Client (and thus lower Manager's realm) now has write permissions.
GetExclusiveAck	Response by paired Client to complete previous GetExclusive request. Signifies paired Client (and thus lower Manager's realm) now has write permissions.
GetEvictAck	Response by paired Client to complete previous GetEvict request. Signifies paired Client has become invalid. Therefore, Manager's realm can safely eliminate all local copies of the block.

Upper Tier Client to Lower Paired Manager *Demand*

Supply	Demand data supply from lower tier's paired Manager or L1. No additional actions required by lower tier.
	Used for data forwarding to satisfy remote read when Manager-Client pair permission levels already match.
Invalidate	Demand lower realm to forfeit write permissions and read permissions, invalidating all local copies of data.
	Used to satisfy remote write request which requires exclusive rights when remote realm already has a copy of the data.
SupplyDowngrade	Demand Data from lower realm's paired Manager. Additionally, lower realm must forfeit write permissions but can retain read permissions and data.
	Used for data forwarding to satisfy remote read when upper-tier paired Client state is forfeiting exclusive/write permissions.
SupplyInvalidate	Demand Data from lower realm's paired manager. Additionally, lower realm must forfeit write permissions AND read permissions, invalidating all local copies of data.
	Used for data forwarding to satisfy remote exclusive/write request when remote realm expects data supplied from this realm.

Lower Tier Manager to Upper Paired Client Demand Reply

SupplyAck	Response by paired Manager to complete previous Supply demand. Supplies data packet.
InvalidateAck	Response by paired Manager to complete previous Invalidate demand. Signifies realm invalidation has completed.
SupplyDowngradeAck	Response by paired Manager to complete previous SupplyDowngrade demand. Supplies data packet and signifies realm downgrade has completed.
SupplyInvalidateAck	Response by paired Manager to complete previous SupplyInvalidate demand. Supplies data packet and signifies realm invalidation has completed.

Table A.1. MCP Actions. Reproduced from Beu, Rosier and Conte [5]

APPENDIX B

DESIGN DIAGRAMS

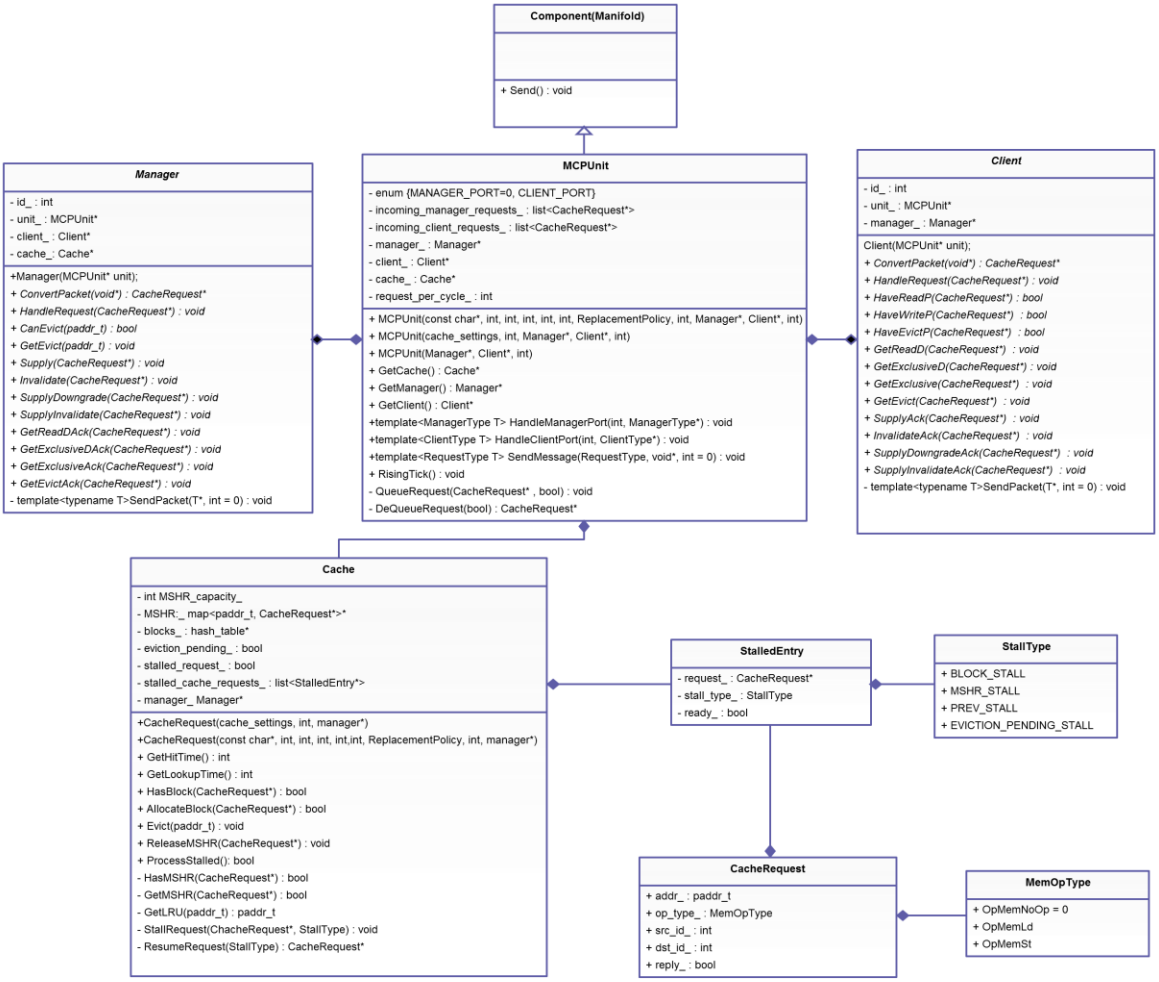

Figure B.1. Class Diagram for the Cache Module. Some Manifold Specific Elements Are Not Shown.

APPENDIX C

CODE LISTINGS

This appendix contains the header files of the five main classes in the cache module: MCPUnit, Cache, Manager, Client and CacheRequest. Note that Manifold specific code might not be included in all files.

```
#ifndef MANIFOLD_MCP_CACHE_MCP_UNIT_H
#define MANIFOLD_MCP_CACHE_MCP_UNIT_H
#include "cache.h"
#include "client.h"
#include "manager.h"
#include "component.h"

namespace manifold {
namespace mcp_cache_namespace {

class MCPUnit : Component
{
public:
    enum {MANAGER_PORT=0, CLIENT_PORT};
    MCPUnit(const char *name, int cache_size, int assoc, int block_size, int hit_time,
                int lookup_time, ReplacementPolicy rp, int mshr_size, Manager* manager, Client* client, int req_per_cycle);
    MCPUnit(cache_settings settings, int mshr_size, Manager* manager, Client* client, int req_per_cycle);
    MCPUnit(Manager* manager, Client* client, int req_per_cycle);
    ~MCPUnit();
    /* Receives a packet from another component and queues it to be processed by the Client */
    template<typename ClientType>  void HandleClientPort(int port, ClientType* message);
    /* Receives a packet from another component and queues it to be processed by the Manager */
    template<typename ManagerType> void HandleManagerPort(int port, ManagerType* message);
    /* Allows the Manager and Client to send messages to other Manifold components through the MCP unit */
    template<typename T> void SendMessage(T* request, void* entity, int delay = 0);
    /* Called every rising edge */
    void RisingTick();
    /* Returns a pointer to the associated Manager */
    Manager* GetManager();
    /* Returns a pointer to the associated Client */
    Client* GetClient();
    /* Returns a pointer to the associated Cache */
    Cache* GetCache();

private:

    /* Number of requests to be processed per tick */
    int requests_per_cycle_;
    /* Deleted by the MCPUnit */
    Manager* manager_;
    /* Deleted by the MCPUnit */
    Client* client_;
    /* Handles blocks, mshrs and stalled requests */
    Cache* cache_;
    /* Holds requests to be processed by the Manager */
    list<CacheRequest*> incoming_manager_requests_;
    /* Holds requests to be processed by the Client */
    list<CacheRequest*> incoming_client_requests_;
    /* Queues an incoming request in the Manager que if for_manager is true or in the Client queue if for_manager is false */
    void QueueRequest(CacheRequest* req, bool for_manager);
    /* Returns a pointer to a de-queued request from the Manager queue if from_manager is true or from the Client queue if from_manager is false.
    Returns a NULL pointer if there are no requests to be de-queued */
    CacheRequest* DeQueueRequest(bool from_manager);
};

} // namespace mcp_cache_namespace
} // namespace manifold

#endif // MANIFOLD_MCP_CACHE_MCP_UNIT_H
```

Figure C.1. MCPUnit Class Header File.

```
#ifndef MANIFOLD_MCP_CACHE_CACHE__H
#define MANIFOLD_MCP_CACHE_CACHE__H

#include <vector>
#include "manager.h"
#include "hash_table.h"
#include "entries.h"

using namespace std;

namespace manifold {
namespace mcp_cache_namespace {

/* Handles  blocks, mshrs and stalled requests */
class Cache
{
public:
    Cache(const char *name, int cache_size, int assoc, int block_size, int hit_time,
                  int lookup_time, ReplacementPolicy rp, int mshr_size);
    Cache(cache_settings settings, int mshr_size);
    ~Cache();
    /* Return the hit time for the cache */
    int GetHitTime();
    /* Returns the look up time for the cache */
    int GetLookUpTime();
    /* Returns true if the block for the request's address is present in the cache */
    bool HasBlock(CacheRequest* req);
    /* Reserves a block for the request's address in the cache. Returns true if the block is obtained or false if the cache is full (Could not evict) */
    bool AllocateBlock(CacheRequest* req);
    /* Evicts the block for the specified address */
    void Evict(paddr_t addr);
    /* Releses the MSHR for the specified address (in MSHR_) */
    void ReleaseMSHR(CacheRequest* addr);
    /* Processes the next request that can be unstalled. If no such request exist returns false */
    bool ProcessStalled();

private:
    /* Maximum Nunmber of MSHRs available to be used */
    int MSHR_capacity_;
    /* MSHRs. At most 1 MSHR per block address */
    map<paddr_t,CacheRequest*> MSHR_;
    /* Cache Storage. Holds the tags of the currently stored blocks */
    hash_table* blocks_;
    /* Signals that an eviction is pending. Another eviction might not be started until the current one completes */
    bool eviction_pending_;
    /* Signals that the cache is processing a previously stalled request */
    bool stalled_request_;
    /* Stalled Cache Requests. Holds a queue of stalled cache requests waiting for blocks, MSHRs or previously stalled requests*/
    list<StalledEntry*> stalled_cache_requests_;
    /* The Manager of the Cache */
    Manager* manager_;

    /* Returns true if there is an MSHR already allocated (in MSHR_) for the specified address */
    bool HasMSHR(CacheRequest* req);
    /* Reserves an MSHR for the specified request (in MSHR_). Returns true if the operation succeeded and false otherwise */
    bool GetMSHR(CacheRequest* req);
    /* Returns the block address of the LRU block */
    paddr_t GetLRU(paddr_t addr);
    /* Queues a CacheRequest waiting for storage, previous requests, or permissions */
    void StallRequest(CacheRequest* request, StallType StallReason);
    /* Returns the first request in the stall queue with the specified StallType */
    CacheRequest* UnStallRequest(StallType ResumeReason);
    /* Returns true if there is a request already stalled for the given address */
    bool HasStall(CacheRequest* request);
    /* Resumes the first request with the specified StalType that will not Stall again */
    void ResumeRequest(StallType ResumeReason);
};

} // namespace mcp_cache_namespace
} // namespace manifold

#endif // MANIFOLD_MCP_CACHE_CACHE_BUFFER_H
```

Figure C.2. Cache Class Header File.

```
#ifndef MANIFOLD_MCP_CACHE_MANAGER_H
#define MANIFOLD_MCP_CACHE_MANAGER_H

#include "cache_request.h"

namespace manifold {
namespace mcp_cache_namespace {

class MCPUnit;
class Client;

class Manager
{
public:
        Manager(MCPUnit* unit);
        ~Manager();
        /* Converts a Manager-protocol packet into a CacheRequest */
        virtual CacheRequest* ConvertPacket(void* packet) = 0;
        /* Handles an incoming CacheRequest */
        virtual void HandleRequest(CacheRequest* req) = 0;
        /* Called by the associated Cache to query for eviction permission */
        virtual bool CanEvict(paddr_t addr) = 0;
        /* Called by the associated Cache to request eviction permission */
        virtual void GetEvict(paddr_t addr)= 0;
        /** Supply data to paired Client. Client expects SupplyAck upon completion */
        virtual void Supply(CacheRequest* req) = 0;
        /** Forfeit both read and write permissions, invaliding all local copies of data. Client expects InvalidateAck upon completion */
        virtual void Invalidate(CacheRequest* req) = 0;
        /**Forfeit write permissions but can retain read permissions and data. Client expects SupplyDowngradeAck upon completion  */
        virtual void SupplyDowngrade(CacheRequest* req) = 0;
        /** Forfeit write permissions and read permissions, invalidating all local copies of data. Client expects SupplyInvalidateAck upon completion */
        virtual void SupplyInvalidate(CacheRequest* req) = 0;
        /** Called from the client to complete previous GetReadD request.  Supplies data packed and signifies paired client now has read permissions */
        virtual void GetReadDAck(CacheRequest* req) = 0;
        /** Called from the client to complete previous GetWrite/GetWriteD request. Supplies data packet and signifies paired client now has write permissions */
        virtual void GetExclusiveDAck(CacheRequest* req) = 0;
        /** Called from the client to complete previous GetWrite request. Supplies data packet and signifies paired client now has write permissions */
        virtual void GetExclusiveAck(CacheRequest* req) = 0;
        /** Called from the client to complete previous GetEvict request. Supplies data packet and signifies paired client is now invalid */
        virtual void GetEvictAck(CacheRequest* req) = 0;
private:

    /* Manager id */
    int id_;
    /* Pointer to the containing unit */
    MCPUnit* unit_;
    /* Pointer to paired client (upper realm) */
    Client* client_;
    /* Pointer to the Cache holding the data */
    Cache* cache_;
    /* Sends a packet on the Manager port */
    template<typename ManagerType> void SendPacket(ManagerType* packet, int delay = 0);
};

} // namespace mcp_cache_namespace
} // namespace manifold

#endif // MANIFOLD_MCP_CACHE_MANAGER_H
```

Figure C.3. Manager Class Header File.

```
#ifndef MANIFOLD_MCP_CACHE_CLIENT_H
#define MANIFOLD_MCP_CACHE_CLIENT_H
#include "cache_request.h"

namespace manifold {
namespace mcp_cache_namespace {

class MCPUnit;
class Manager;
class Cache;

class Client
{
public:
        Client(MCPUnit* unit);
        ~Client();

        /* Converts a Client-protocol packet into a CacheRequest */
        virtual CacheRequest* ConvertPacket(void* packet) = 0;
        /* Handles an incoming CacheRequest */
        virtual void HandleRequest(CacheRequest* req) = 0;

        /** Returns true if you have read permissions */
        virtual bool HaveReadP(CacheRequest* req) = 0;
        /** Returns true if you have write permissions */
        virtual bool HaveWriteP(CacheRequest* req) = 0;
        /** Returns true if you can be safely evicted */
        virtual bool HaveEvictP(CacheRequest* req) = 0;
        /** Client begins data and read permission acquistion within its native coherence realm. Lower manager expects GrantReadD upon completion */
        virtual void GetReadD(CacheRequest* req) = 0;
        /** Client begins data and write permission acquistion within its native coherence realm Lower anager expects GrantWriteD upon completion */
        virtual void GetExclusiveD(CacheRequest* req) = 0;
        /** Client begins write permission acquistion within its native coherence realm. Lower manager expects GrantWrite or GrantWriteD upon completion */
        virtual void GetExclusive(CacheRequest* req) = 0;
        /** Client begins eviction sequence within its native coherence realm. Lower manager expects GetEvictAck upon completion */
        virtual void GetEvict(CacheRequest* req) = 0;
        /** Called by the Manager to complete previous Supply demand. Supplies data packet */
        virtual void SupplyAck(CacheRequest* req) = 0;
        /** Called by the Manager to complete previous Invalidate demand.  Signifies realm invalidation has completed */
        virtual void InvalidateAck(CacheRequest* req) = 0;
        /** Called by the Manager to complete preivous SupplyDowngrade demand.  Supplies data packet and signifies realm downgrade has completed */
        virtual void SupplyDowngradeAck(CacheRequest* req) = 0;
        /** Called by the Manager to complete preivous SupplyInvalidate demand.  Supplies data packet and signifies realm invalidation has completed */
        virtual void SupplyInvalidateAck(CacheRequest* req) = 0;

private:
        /* Client id */
        int id_;
        /* Pointer to containing unit */
        MCPUnit* unit_;
        /* Pointer to paired manager (lower realm) */
        Manager* manager_;
        /* Sends a packet on the Client port */
        template<typename ClientType> void SendPacket(ClientType* packet, int delay = 0);
};

} // namespace mcp_cache_namespace
} // namespace manifold
#endif // MANIFOLD_MCP_CACHE_CLIENT_H
```

Figure C.4. Client Class Header File.

```
#ifndef MANIFOLD_MCPCACHE_CACHE_REQUEST_H
#define MANIFOLD_MCPCACHE_CACHE_REQUEST_H

#include "definitions.h"
namespace manifold {
namespace mcp_cache_namespace {

enum MemOpType {
    OpMemNoOp = 0,
    OpMemLd,
    OpMemSt,
};


class CacheRequest {
    public:
        paddr_t addr_;
        MemOpType op_type_;
        int src_id_;
        int dst_id_;
        bool reply_;
        CacheRequest();  //for deserialization
};

} // namespace mcp_cache_namespace
} // namespace manifold

#endif // MANIFOLD_MCPCACHE_CACHE_REQUEST_H
```

Figure C.5. CacheRequest Class Header File.

REFERENCES

[1] www.manifold.gatech.edu

[2] J. Wang, J. G. Beu, S. Yalamanchili, and T. M. Conte. “Designing Configurable, “Modifiable and Reusable Components for Simulation of Multicore Systems.” *3rd International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems* (PMBS12), November 2012.

[3] "Manifold 0.9 Simulation Kernel API http://manifold.gatech.edu/wp-content/uploads/2013/02/kernel_api_0.9.pdf

[4] C. Kersey, A. Rodrigues, and S. Yalamanchili. “A Universal Parallel Front-End for Execution-Driven Microarchitecture Simulation.” *HIPEAC Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, January 2012.

[5] J. G. Beu, M. C. Rosier and T. M. Conte, “Manager-Client Pairing: A Framework for Implementing Coherence Hierarchies,” *Proceedings of the 44th Annual International Symposium on Microarchitecture* (MICRO-44), (Porto Alegre, Brazil), Dec., 2011.

[6] J. G. Beu, Jason A. Poovey, Eric R. Hein, T. M. Conte, “High-Speed Formal Verification of Heterogeneous Coherence Hierarchies,” *The 19th IEEE International Symposium on High Performance Computer Architecture* (HPCA'13), (Shenzhen, China), Feb., 2013.

[7] “Manifold 0.9 Component Developer's Guide” http://manifold.gatech.edu/wp-content/uploads/2013/02/component_developer_guide_0.9.pdf

[8] David Kroft. 1981. “Lockup-free instruction fetch/prefetch cache organization.” *In Proceedings of the 8th annual symposium on Computer Architecture* (ISCA '81). IEEE Computer Society Press, Los Alamitos, CA, USA, 81-87.

[9] Adve, S.V.; Gharachorloo, K., "Shared memory consistency models: a tutorial," *Computer*, vol.29, no.12, pp.66, 76, Dec 1996

[10] K. I. Farkas and N. P. Jouppi. 1994. Complexity/performance tradeoffs with non-blocking loads. *In Proceedings of the 21st annual international symposium on*

Computer Architecture (ISCA '94). IEEE Computer Society Press, Los Alamitos, CA, USA, 211-222.